A *really* simple approximation of smallest grammar

Artur Jeż

Max Planck Institute for Informatics

Moscow, 18.06.2014

Compression and grammars

Compression

- Increasingly popular

Compression and grammars

Compression

- Increasingly popular

Grammars based compression

- CFG defining unique word
- Straight Line Programs (SLP)

Compression and grammars

Compression

- Increasingly popular

Grammars based compression

- CFG defining unique word
- Straight Line Programs (SLP)
- easy to work on
- related to block compression

Smallest grammar

Problem

Given w return smallest CFG G_w such that $L(G_w) = w$.

Smallest grammar

Problem

Given w return smallest CFG G_w such that $L(G_w) = w$.

NP-hard.

Best approximation ratio

$\mathcal{O}(\log(n/g))$, where g is the size of the optimal grammar.

Smallest grammar

Problem

Given w return smallest CFG G_w such that $L(G_w) = w$.

NP-hard.

Best approximation ratio

$\mathcal{O}(\log(n/g))$, where g is the size of the optimal grammar.

This talk

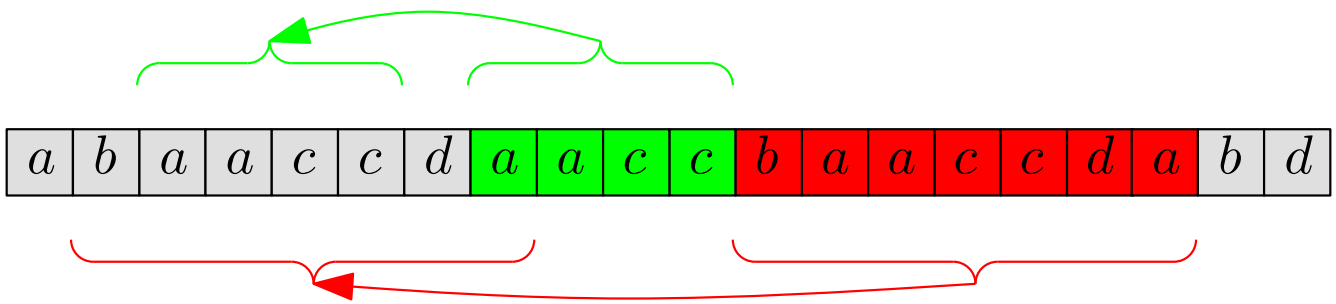
A really simple linear algorithm with this bound.

LZ77

- Represent w as $w = f_1 f_2 f_3 \dots f_\ell$.
- Each $f = w[i \,..\, i + |f|)$ is

  free letter a letter *or*

  factor equal to $w[j \,..\, j + |f|)$ for some $j < i$.
- size of factorisation $f_1 f_2 f_3 \dots f_\ell$ is ℓ

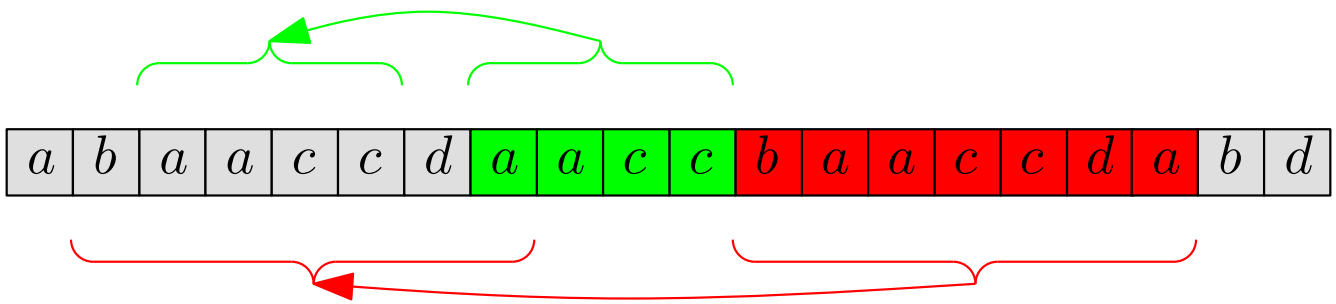


LZ77

- Represent w as $w = f_1 f_2 f_3 \dots f_\ell$.
- Each $f = w[i \,..\, i + |f|)$ is

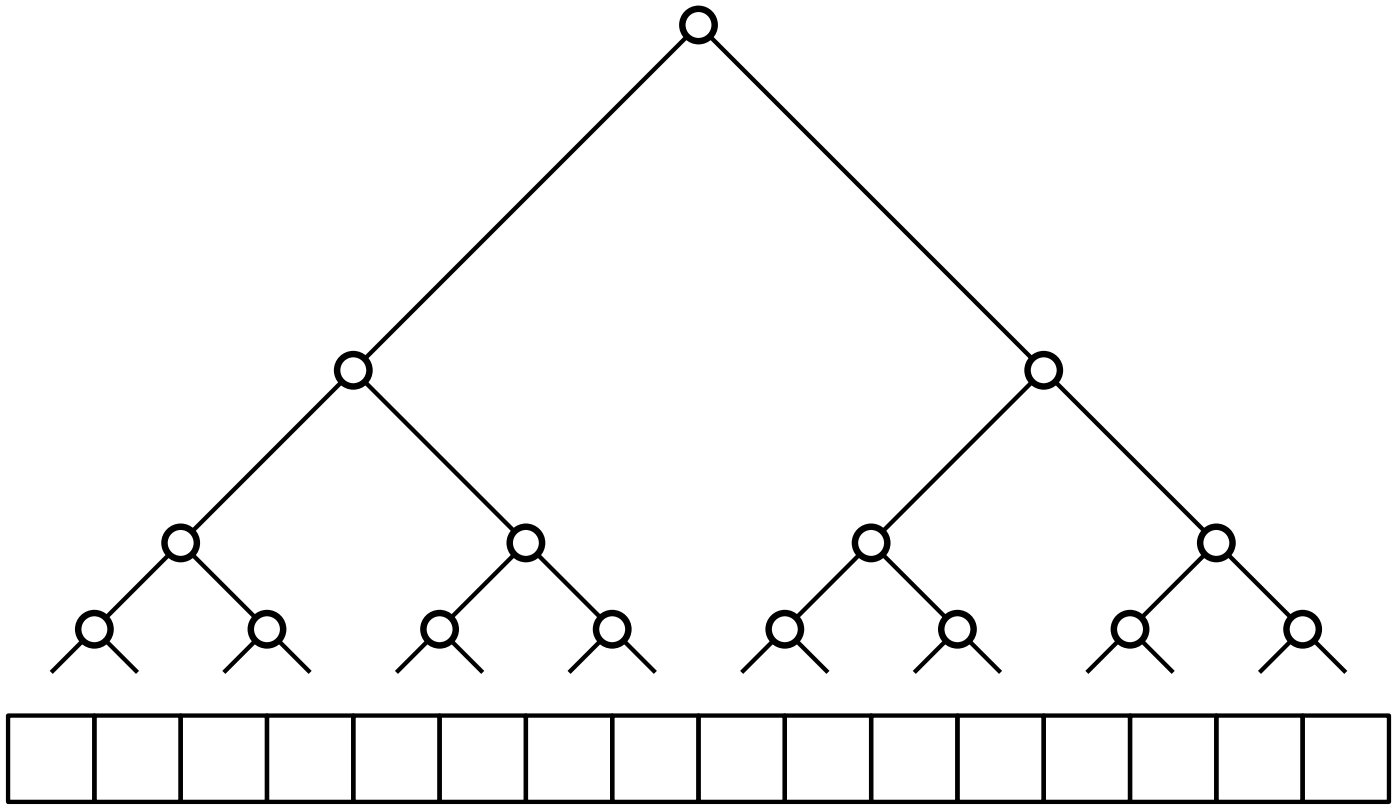
  free letter a letter *or*

  factor equal to $w[j \,..\, j + |f|)$ for some $j < i$.
- size of factorisation $f_1 f_2 f_3 \dots f_\ell$ is ℓ

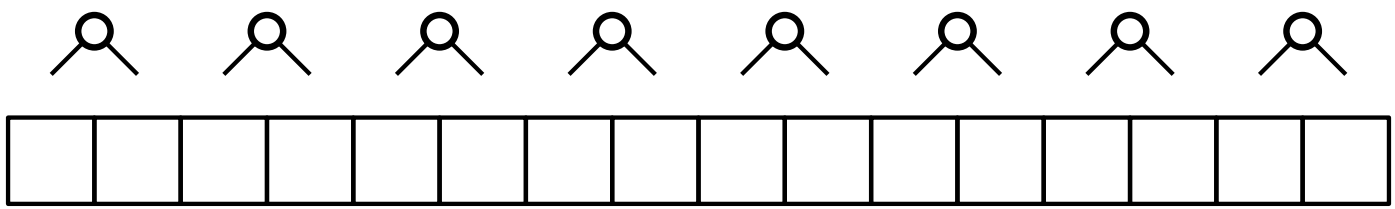


- smallest can be found in $\mathcal{O}(|w|)$
- smaller than smallest grammar

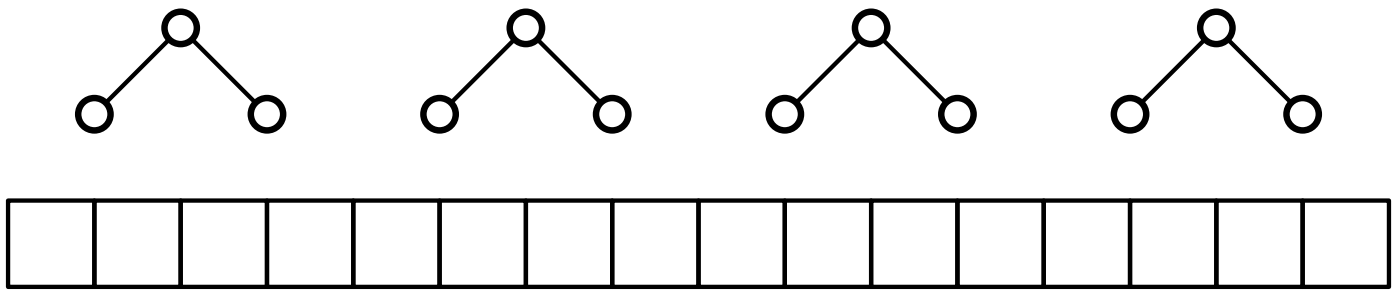
Grammar = iterated pairing



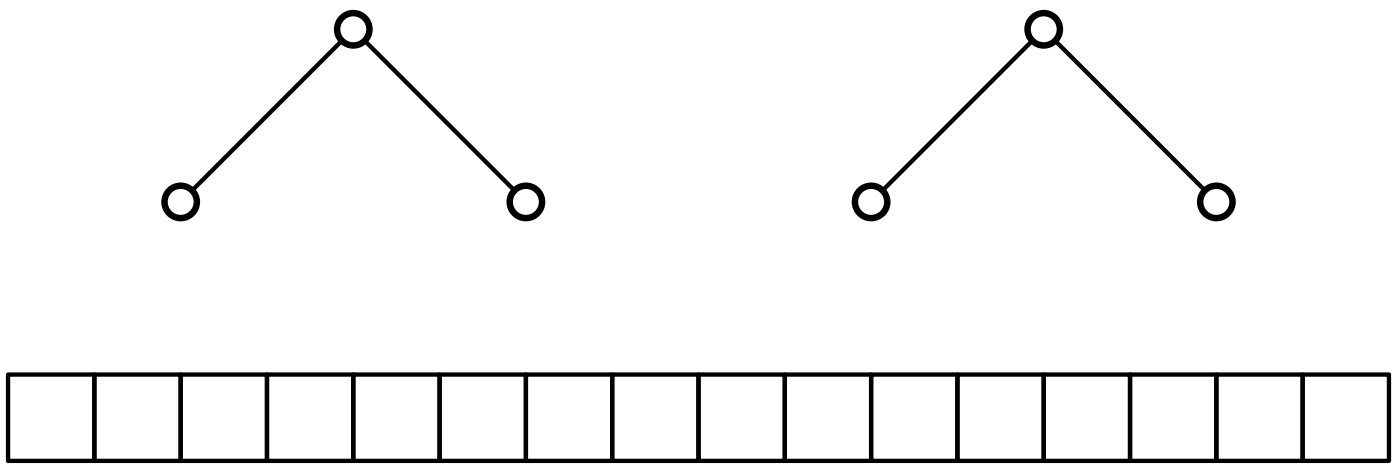
Grammar = iterated pairing



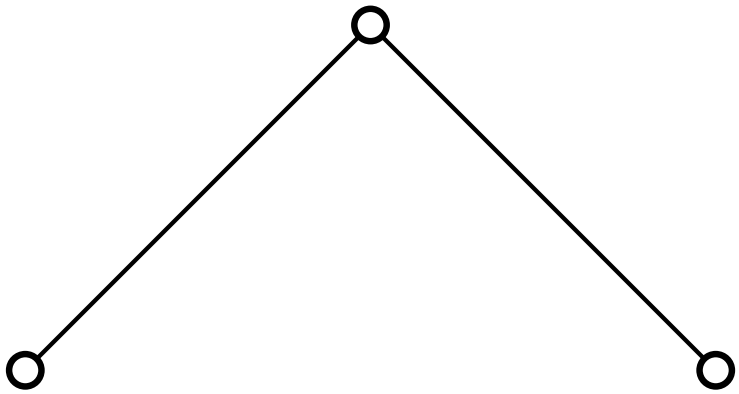
Grammar = iterated pairing

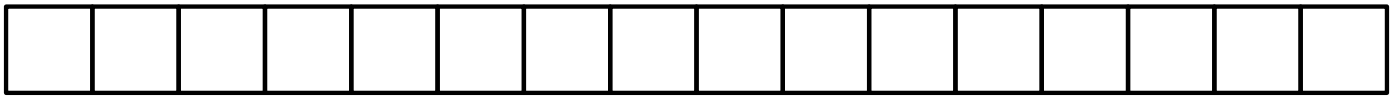


Grammar = iterated pairing

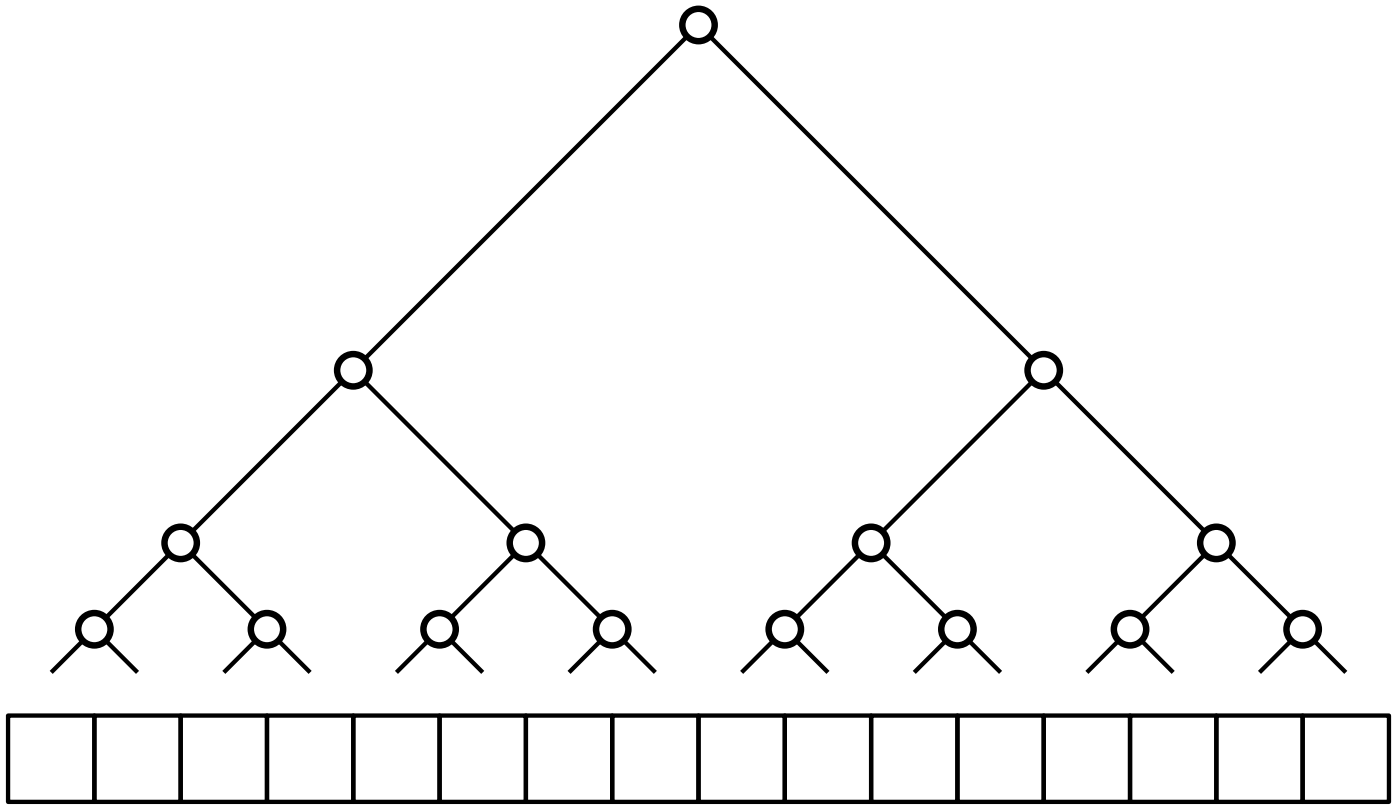


Grammar = iterated pairing



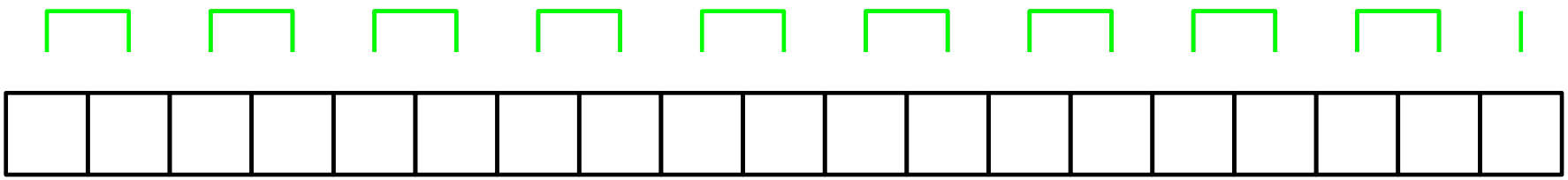


Grammar = iterated pairing

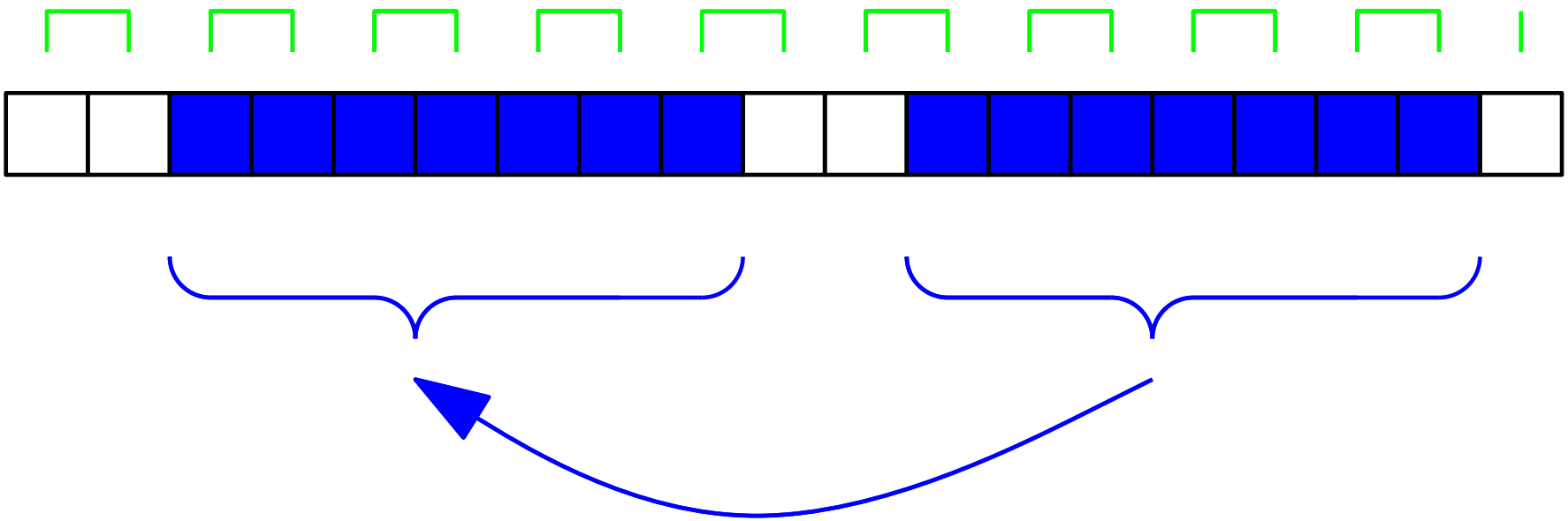


- grammar = iterated pairing
- size = different pairs (in total)

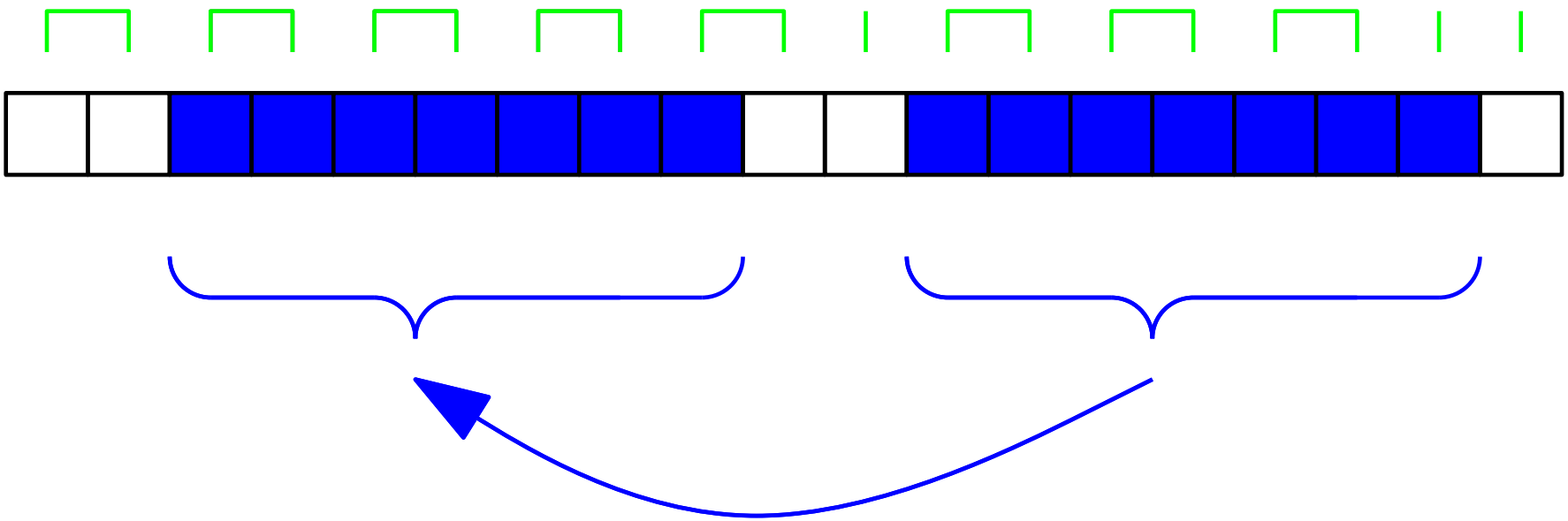
Pairing



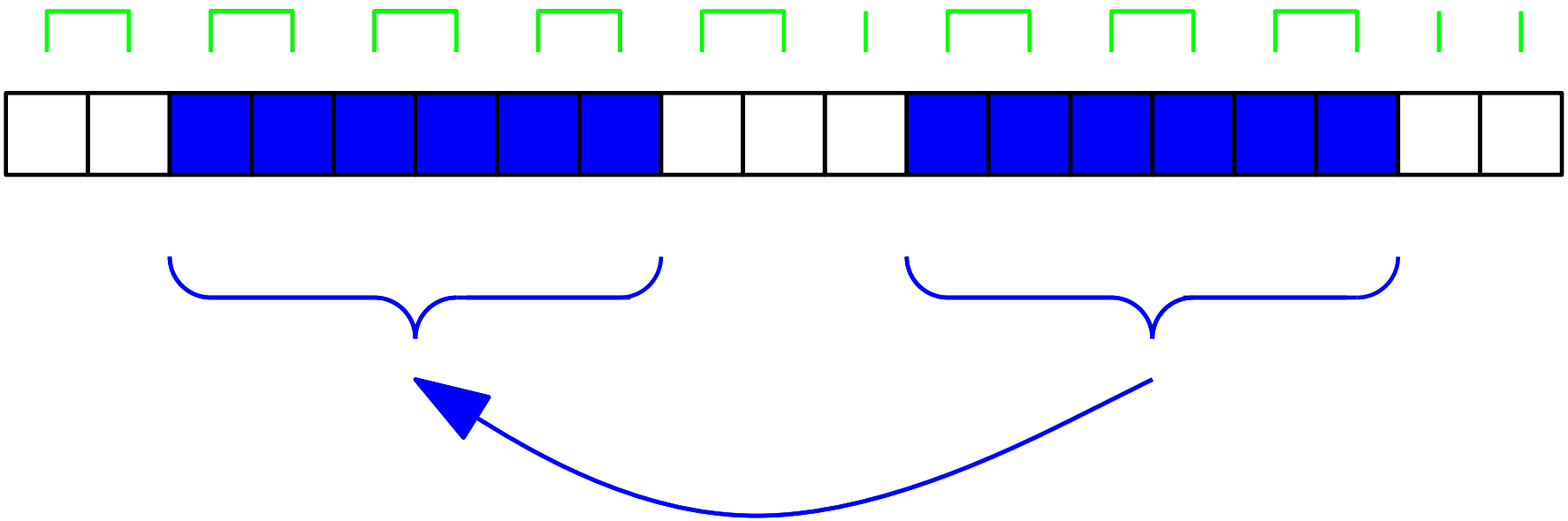
Pairing



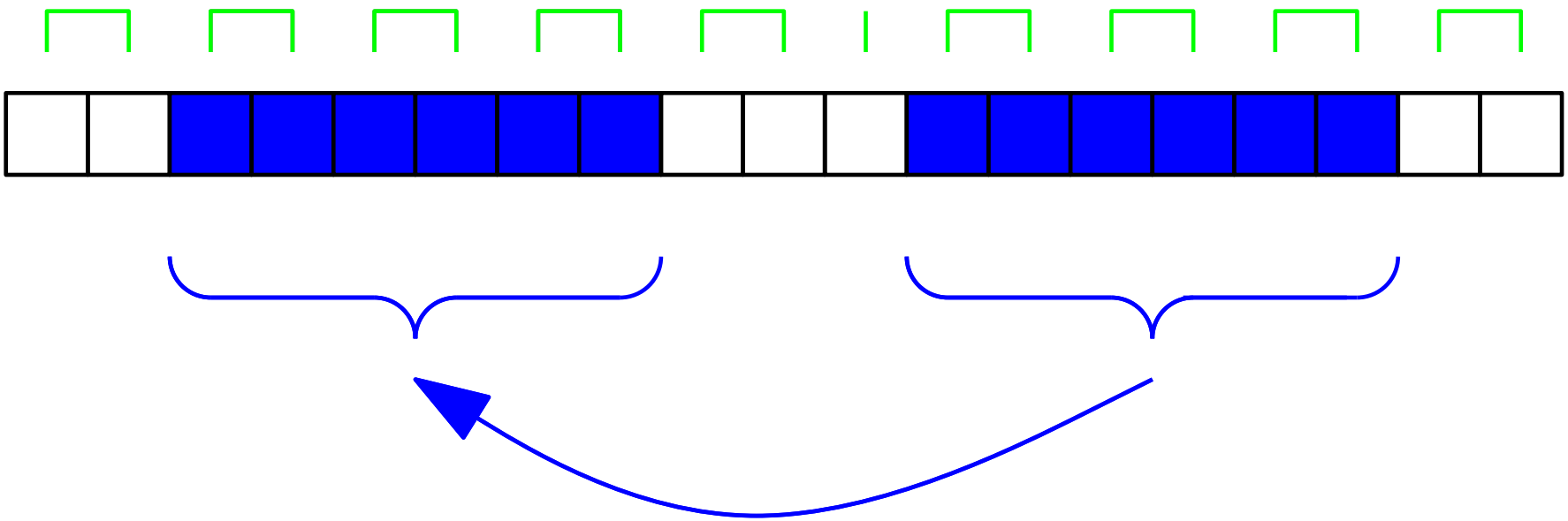
Pairing



Pairing



Pairing



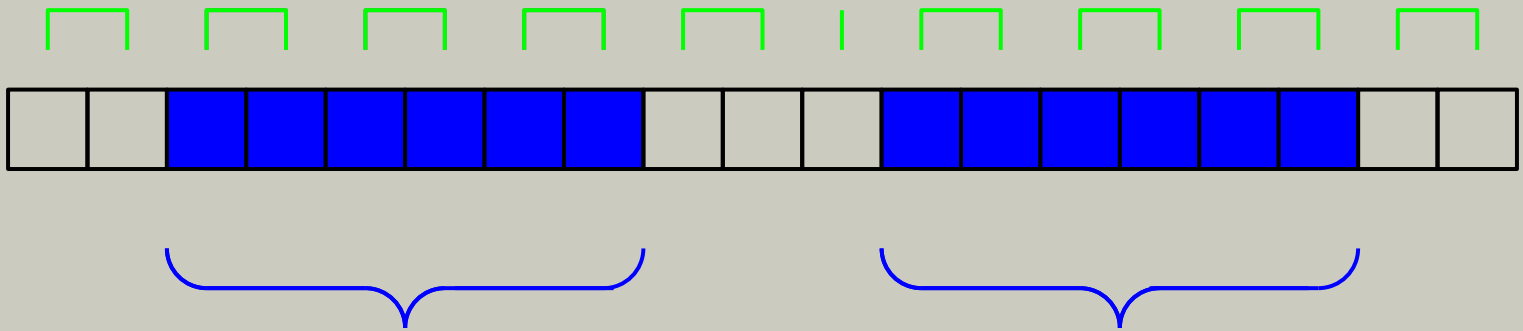
Properties

- no two consecutive letters are unpaired
- factor is paired as its definition
- first two (last two) letters of a factor are paired

Properties

- no two consecutive letters are unpaired
- factor is paired as its definition
- first two (last two) letters of a factor are paired

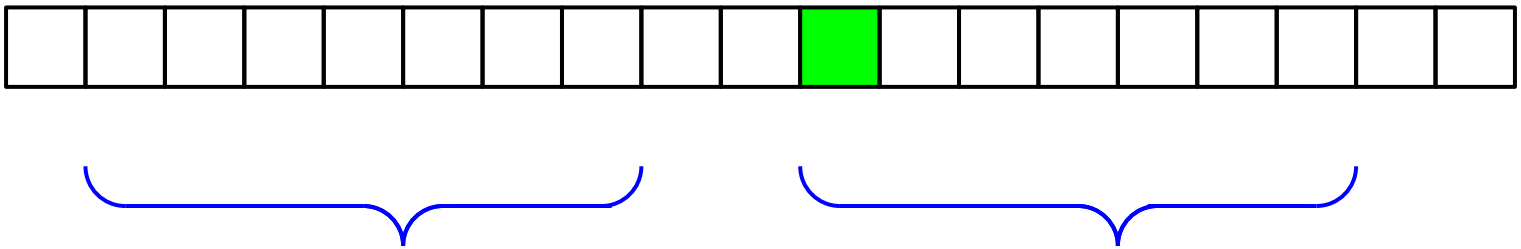
After replacing: factorisation with the same number of factors.



Pairing

1: **while** $i \leq |t|$ **do**

Pairing



1: **while** $i \leq |t|$ **do**
2: **if** i is the first letter of a factor **then**

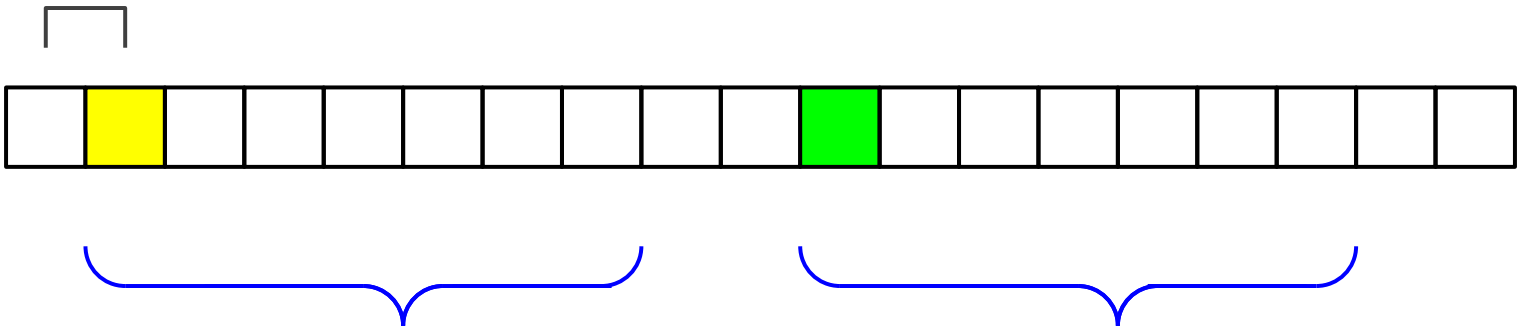
Pairing

1: **while** $i \leq |t|$ **do**
2: **if** i is the first letter of a factor **then**
3: **if** factor has one letter **then**

Pairing

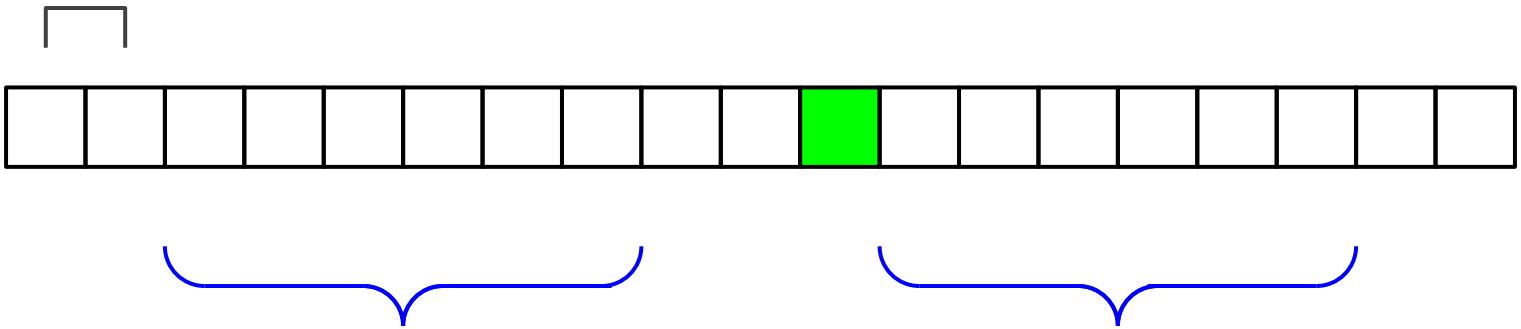
1: **while** $i \leq |t|$ **do**
2: **if** i is the first letter of a factor **then**
3: **if** factor has one letter **then**
4: replace it with a free letter

Pairing



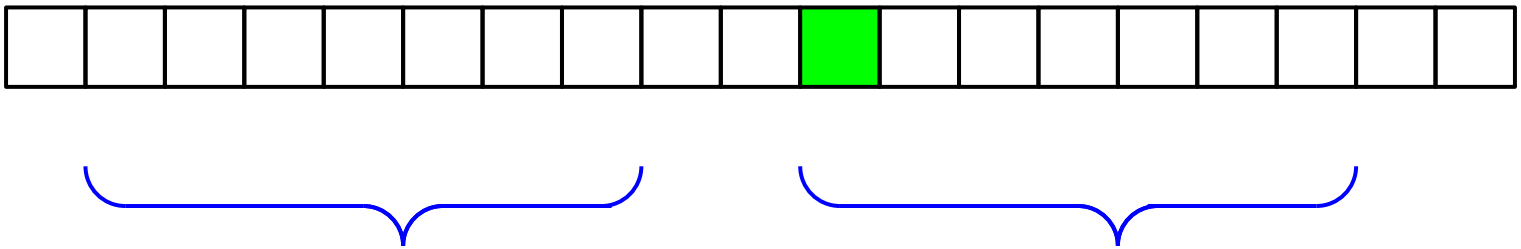
1: **while** $i \leq |t|$ **do**
2: **if** i is the first letter of a factor **then**
3: **if** factor has one letter **then**
4: replace it with a free letter
5: **else if** j is not a first letter in a pair **then**

Pairing



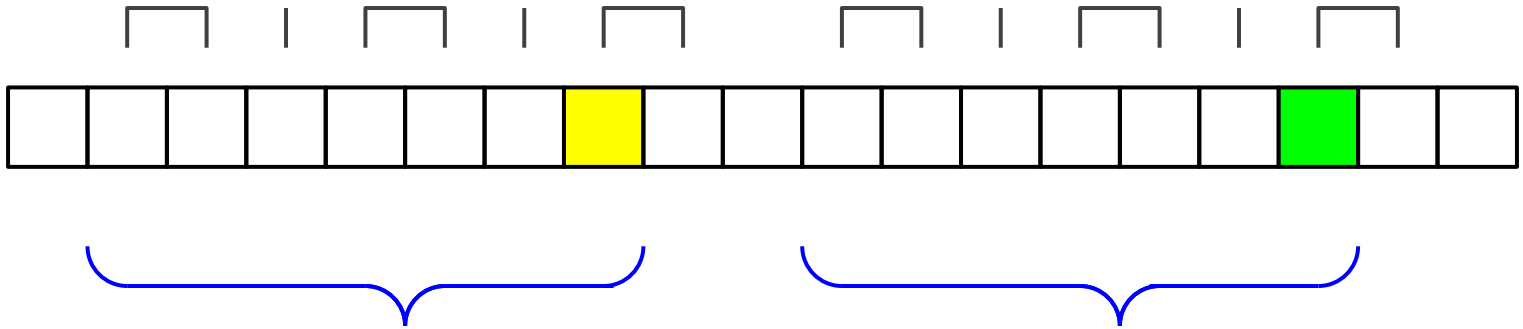
```
1: while i ≤ |t| do
2:     if i is the first letter of a factor then
3:         if factor has one letter then
4:             replace it with a free letter
5:         else if j is not a first letter in a pair then
6:             shorten the factor (from left)
```

Pairing



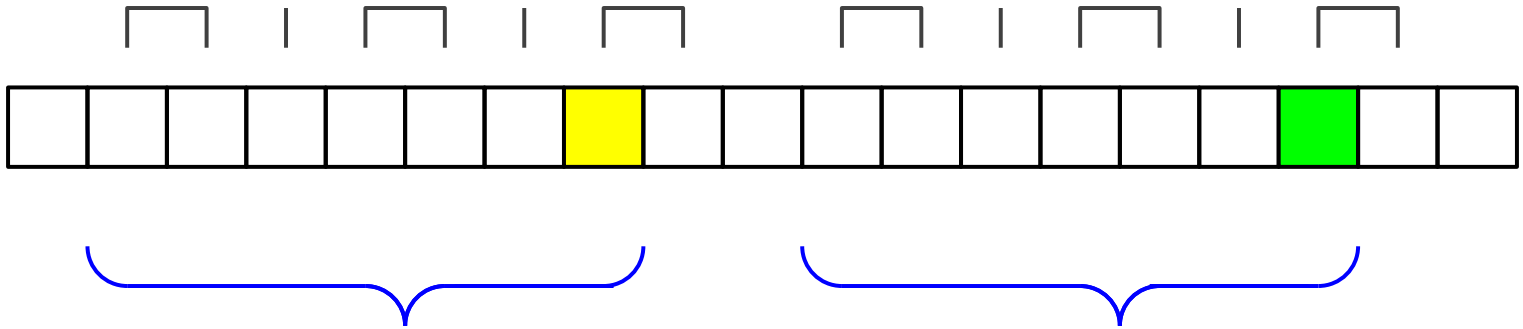
```
1: while i ≤ |t| do
2:     if i is the first letter of a factor then
3:         if factor has one letter then
4:             replace it with a free letter
5:         else if j is not a first letter in a pair then
6:             shorten the factor (from left)
7:         else
8:             copy the pairing for the whole factor (move i)
```

Pairing



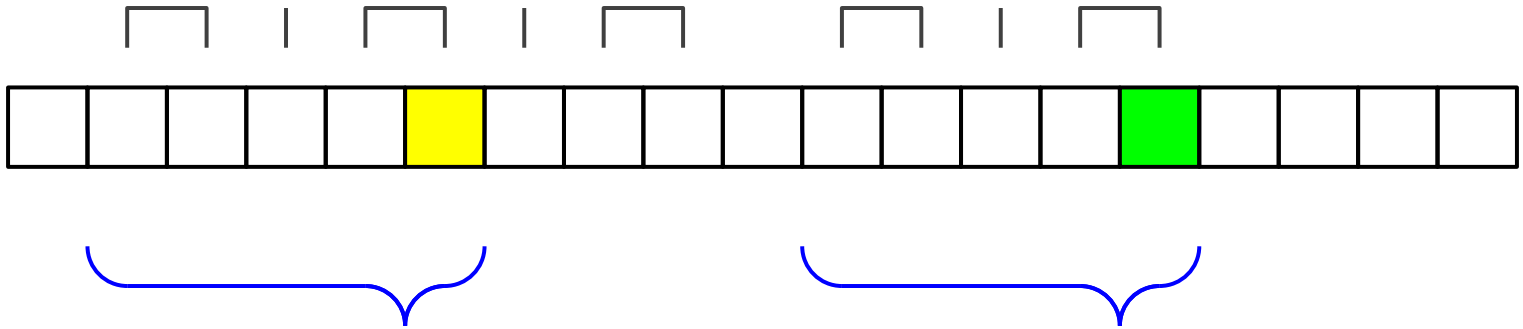
```
1: while i ≤ |t| do
2:     if i is the first letter of a factor then
3:         if factor has one letter then
4:             replace it with a free letter
5:         else if j is not a first letter in a pair then
6:             shorten the factor (from left)
7:         else
8:             copy the pairing for the whole factor (move i)
```

Pairing



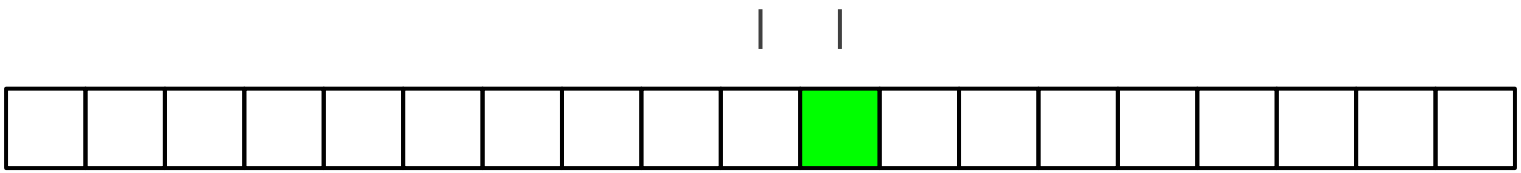
```
1: while i ≤ |t| do
2:     if i is the first letter of a factor then
3:         if factor has one letter then
4:             replace it with a free letter
5:         else if j is not a first letter in a pair then
6:             shorten the factor (from left)
7:         else
8:             copy the pairing for the whole factor (move i)
9:             while j is not a second in a pair do
10:                shorten the factor (from the right)
```

Pairing

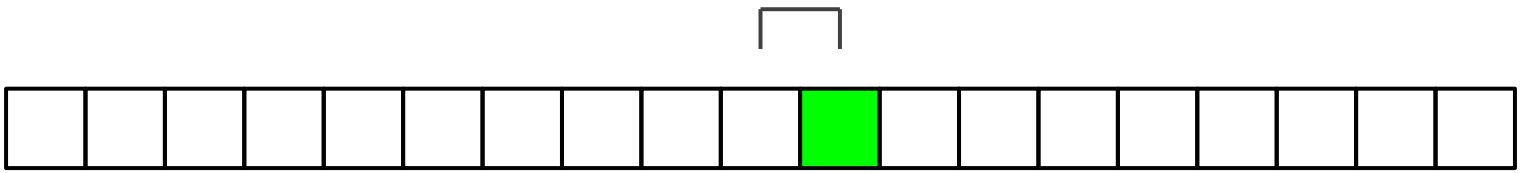


```
1:  while i ≤ |t| do
2:      if i is the first letter of a factor then
3:          if factor has one letter then
4:              replace it with a free letter
5:          else if j is not a first letter in a pair then
6:              shorten the factor (from left)
7:          else
8:              copy the pairing for the whole factor (move i)
9:              while j is not a second in a pair do
10:                 shorten the factor (from the right)
```

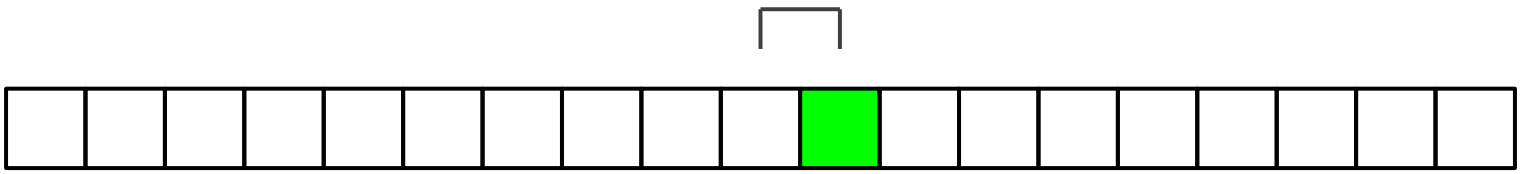
1: **while** $i \leq |t|$ **do**
2: . . .
3: **if** i is a free letter **then**



1: **while** $i \leq |t|$ **do**
2: . . .
3: **if** i is a free letter **then**
4: **if** previous letter unpaired **then**



```
1: while i ≤ |t| do
2:     ...
3:     if i is a free letter then
4:         if previous letter unpaired then
5:             pair them                          ▷ New pair!
```

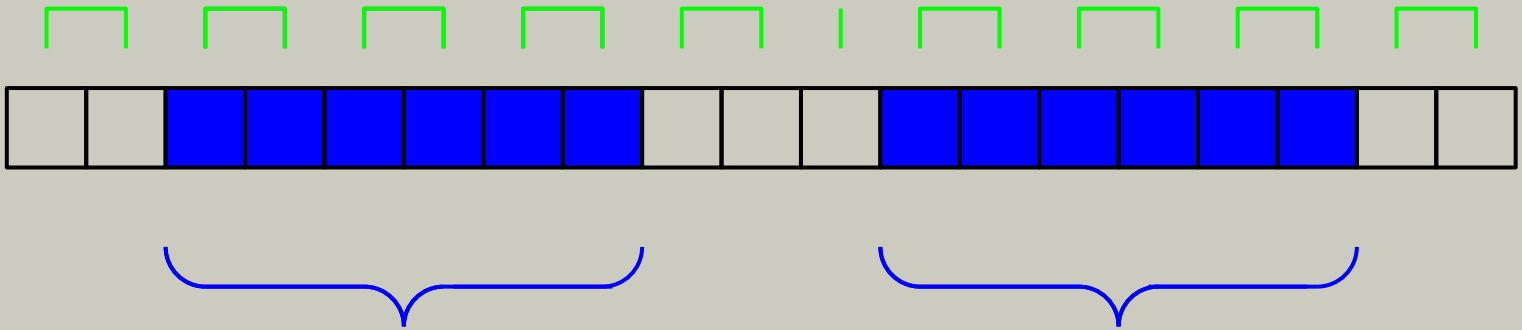


1: **while** $i \leq |t|$ **do**
2: . . .
3: **if** i is a free letter **then**
4: **if** previous letter unpaired **then**
5: pair them ▷ New pair!

- Invariants are preserved

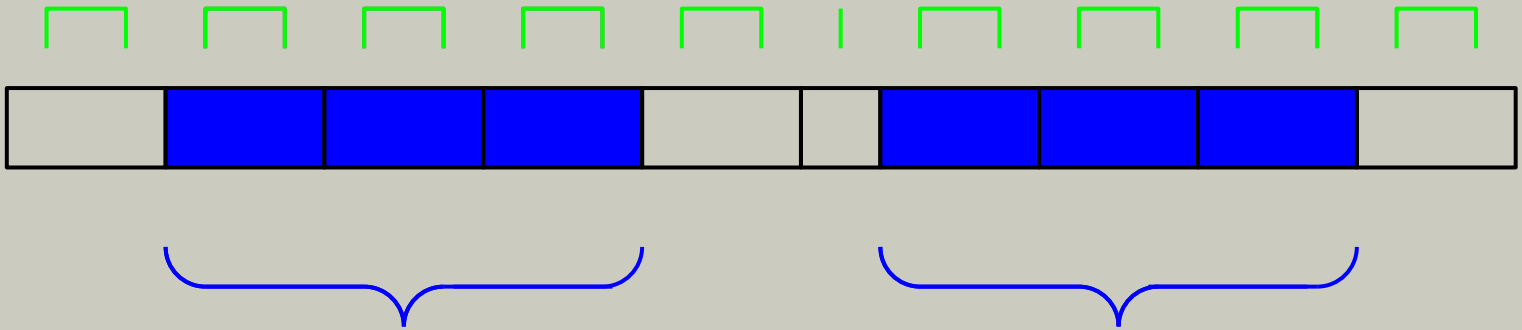
Pair replacement

- replace pairs with new letters
- leave unpaired letters as they are



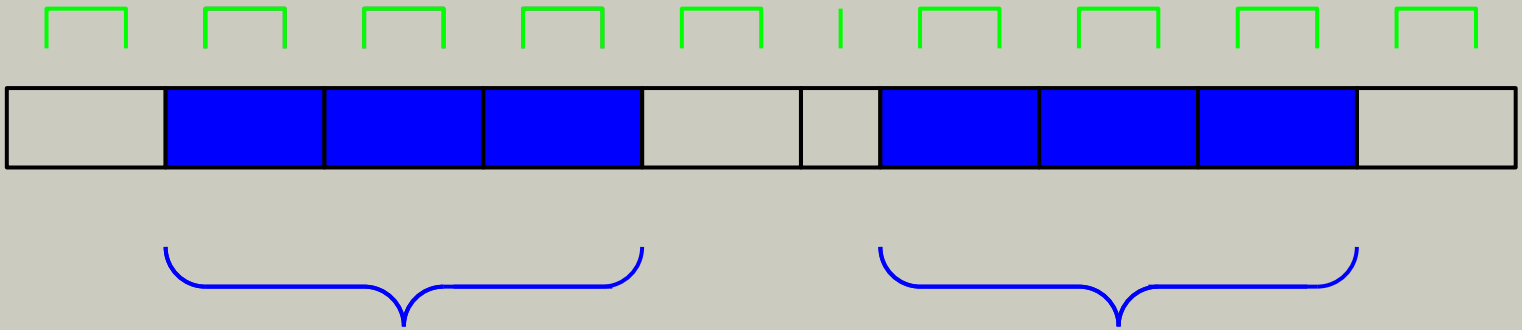
Pair replacement

- replace pairs with new letters
- leave unpaired letters as they are



Pair replacement

- replace pairs with new letters
- leave unpaired letters as they are



- we ‘inherit’ the factorisation
  - factors well-defined: factor and definition paired the same identify old and new
  - free letters: old ones or pairs of old ones

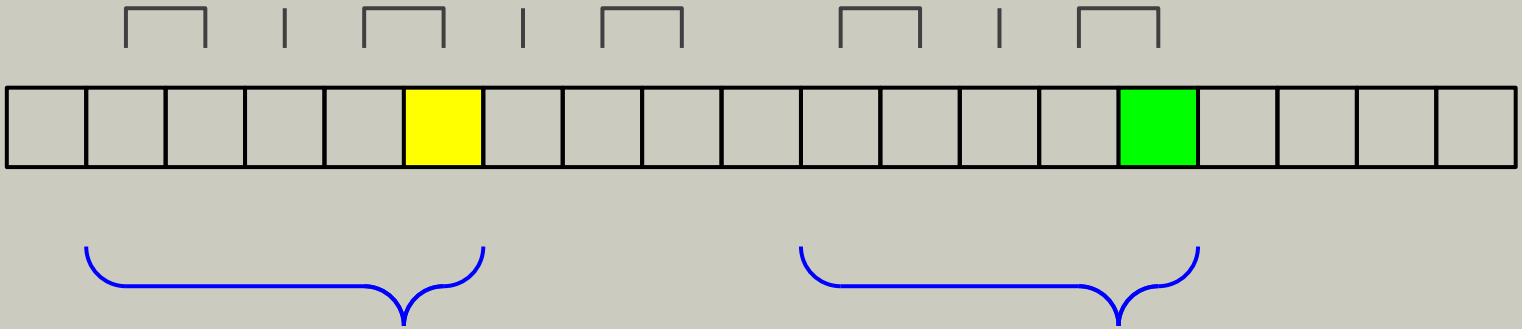
Analysis

Recall: number of nonterminals = number of different pairs

Analysis

Recall: number of nonterminals = number of different pairs

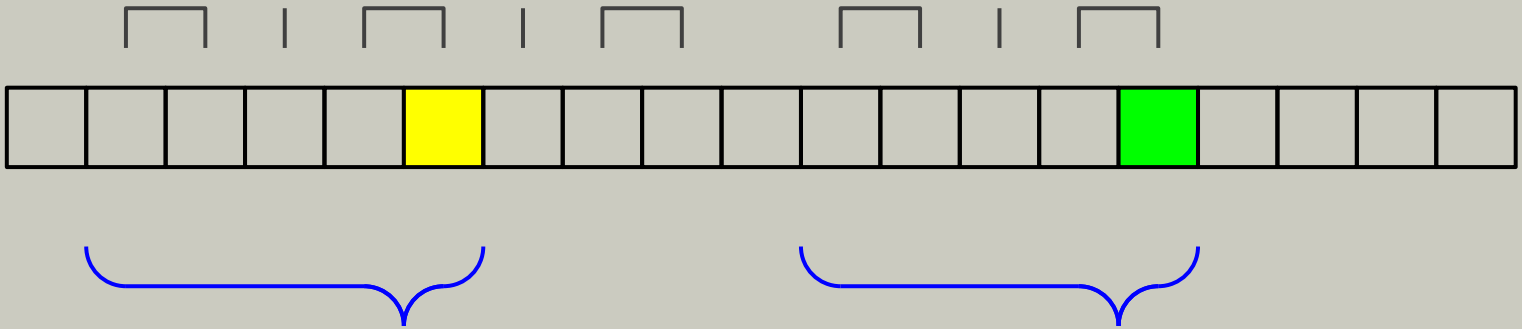
- pairs inside a factor are not new



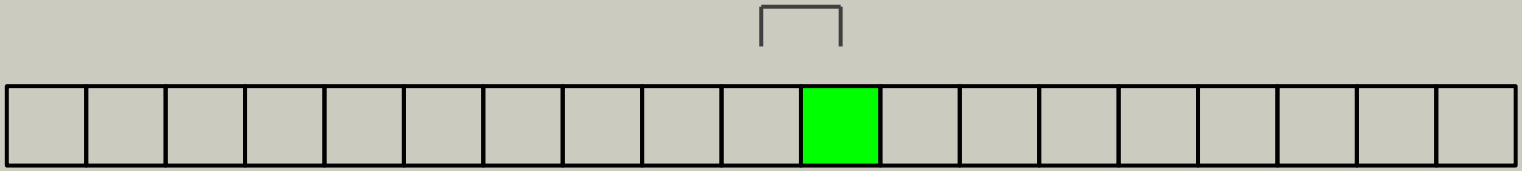
Analysis

Recall: number of nonterminals = number of different pairs

- pairs inside a factor are not new



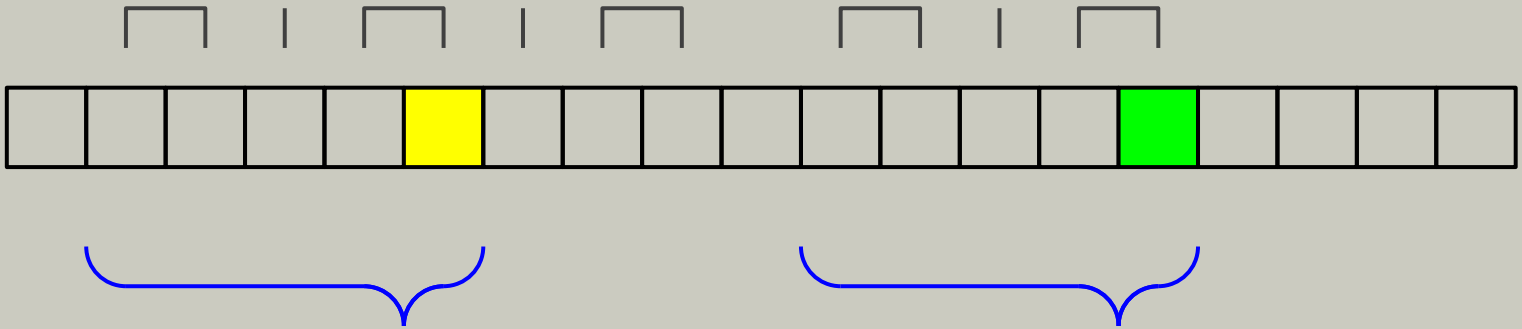
- a pair of free letters creates one new



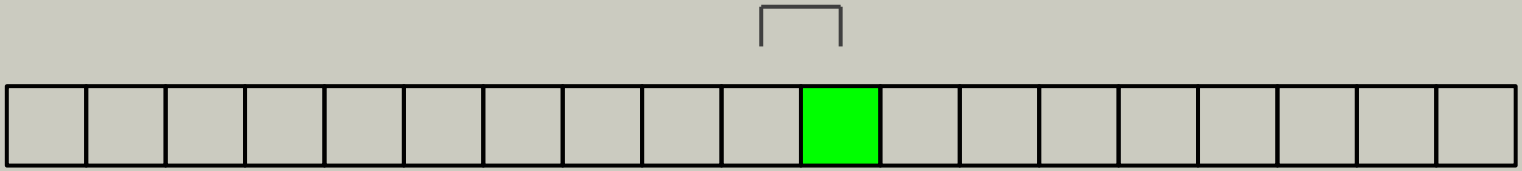
Analysis

Recall: number of nonterminals = number of different pairs

- pairs inside a factor are not new



- a pair of free letters creates one new



- compressed into a single free letter: decrease count by 1
- need to count number of introduced free letters

Analysis continued

Fix a factor and phase.

How many free letters?

- when replaced with a letter: one in total
- two from the left (pop until a first in a pair is found: at most 2)
- two from the right (symmetric)

Analysis continued

Fix a factor and phase.

How many free letters?

- when replaced with a letter: one in total
- two from the left (pop until a first in a pair is found: at most 2)
- two from the right (symmetric)

Factor f is present in $\mathcal{O}(\log |f|)$ rounds

$$\sum_{i=1}^{\ell} \log |f_i| \text{ when } \sum_{i=1}^{\ell} |f_i| = n$$

$$\mathcal{O}(\ell \log(n/\ell)) \leq \mathcal{O}(g \log(n/g))$$

Open problems

- Lower bound
  - only constant lower bound
  - already for very simple strings: $a^{n_1} b a^{n_2} b \cdot b a^{n_k}$
- What is the approximation bound?
- Hardness?
- Simpler subclasses of strings?